Measuring the Usability and Capability of App Inventor to Create Mobile Applications

Benjamin Xie   Isra Shabir   Hal Abelson

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
{bxie, ishabir, hal} @mit.edu

Abstract

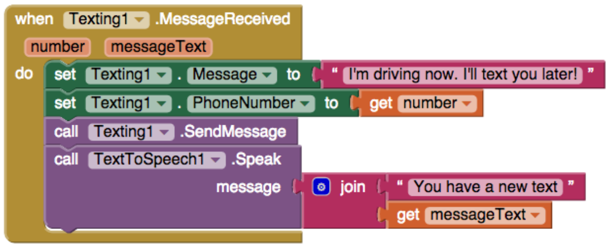
MIT App Inventor is a web service that enables users with little to no previous programming experience to create mobile applications using a visual blocks language. We analyze a sample of 5,228 random projects from the corpus of 9.7 million and group projects by functionality. We then use the number of unique blocks in projects as a metric to better understand the usability and realized capability of using App Inventor to implement specific functionalities. We introduce the notion of a usability score and our results indicate that introductory tutorials heavily influence the usability of App Inventor to implement particular functionalities. Our findings suggest that the sequential nature of App Inventor's learning resources results in users realizing only a portion of App Inventor's capabilities and propose improvements to these learning resources that are transferable to other programming environments or tools with an online presence.

Keywords Mobile Computing, Computer Science Education, Qualitative Study, End-User Programming, Visual Languages

1. Introduction

MIT App Inventor is an environment that leverages a blocks-based visual programming language to enable people to create mobile apps for Android devices [**?**]. An App Inventor project consists of a set of components and a set of program blocks that enable the functionality of these components. Components include items visible on the phone screen (e.g. buttons, text boxes, images, drawing canvas) as well as non-visible items (e.g. camera, database, speech recognizer, GPS location sensor). The app is programmed using Blockly, a visual blocks-based programming framework [**?**]. Figure 1 shows the program blocks for an app to discourage texting while driving. When a text is received, a default message is sent back in response and the received text is read aloud.

There have been two main versions of App Inventor. App Inventor Classic (also known as App Inventor 1) was released in 2009 and ran its blocks editor in a separate Java application. In late 2013, App Inventor 2 (AI2) was released; the blocks editor now runs in a web browser. This research focuses on App Inventor 2 data [**?**].



```
when Texting1 .MessageReceived
  number  messageText
do  set Texting1 . Message to " I'm driving now. I'll text you later! "
    set Texting1 . PhoneNumber to get number
    call Texting1 .SendMessage
    call TextToSpeech1 .Speak
                     message  join " You have a new text "
                                   get messageText
```

Figure 1. Blocks for an App Inventor 2 project that automatically respond to texts received with a predefined message and reads the received text aloud.

App Inventor is taught to a broad audience, ranging from grade school to college students. Reports on courses taught depict App Inventor being used to create very diverse apps. These apps range from apps that discourage texting while driving, to apps that track school buses [**?**], to apps that organize community service cleanups [**?**]. The pattern we observe is that App Inventor enables "situated computing" [**?**]. This quarter-century old concept suggests that the convergence of computing, connectivity, and content enables users to harness computing to bridge the gap between intentions and actions. App Inventor allows people to leverage their mobile devices and solve everyday problems they encounter.

App Inventor also has copious resources for self-learners, typically in the form of self-contained tutorials. A survey of 129,130 self-selected App Inventor users found that 73% of respondents used App Inventor at home, suggesting a significant portion of App Inventor users learn to use the service on their own and not in a formal learning environment. The App Inventor resources page includes 26 tutorials ranging from beginner to advanced difficulty [**?**]. These tutorials involve creating an entire functioning app from start to finish. Each tutorial typically focuses on either introducing a new component (such as a canvas or GPS integration) or additional functionality for a previously introduced component.

To date, over 3.5 million users from 195 countries have created over 9.7 million apps with the MIT App Inventor service [**?**].

2. Objective

The goal of this paper is to evaluate the usability and capability of App Inventor to create apps of differing functionality by analyzing the apps created with App Inventor. We define usability as the ease of use of the App Inventor service to create an app. We define capability as the extent of App Inventor potential that is realized by users to implement certain functionality.

A guiding principle to the creation of a programming environment is the idea of a "low floor, high ceiling" [**?**]. That is, the environment must be usable enough such that beginners can easily create a basic yet functioning program (low floor), but also have extensible capabilities such that advanced users can also benefit (high ceiling). We are particularly interested in comparing the usability and capability of App Inventor for creating apps of differing functionality.

We analyze a random sample of projects and group them based on the types of components used in the app. We then look at both the number of unique blocks in projects. We then evaluate how well suited the App Inventor environment is to creating apps with various functionalities.

In this paper, we explain our technical approach of extracting information from raw project data, filtering and grouping projects, and comparing the grouped projects given our metrics. We then discuss our findings in the context of the App Inventor service and its teaching resources.

3. Related Work

Prior to this work, analysis of App Inventor Classic data has been done by [**?**]. Some of the notable findings:

- Nearly 50% of users do not have a single block or component in any of their projects.
- 30% of all apps have no blocks and are therefore static and have no behavior.
- 51% of procedures are only called 0 or 1 time.

This data indicates that a large number of App Inventor Classic projects were never completed. It was suggested that a major contributing factor is the usability of the service. Whereas App Inventor 2 is a single-page web service, App Inventor Classic required the deployment of an external Java service to program the app. The high proportion of projects without blocks motivated the usability changes of the blocks in App Inventor 2.

An environment that leverages a blocks-based programming language very similar to App Inventor's is Scratch. Scratch enables users to create interactive stories, games, and animations [**?**]. One research study on Scratch examined trends in user participation in Scratch [**?**]. This study categorized the Scratch blocks into five categories (Loops, Booleans, Operator, Broadcasts, and Variables) to illustrate different programming concepts in Scratch. Projects were differentiated according to the number of blocks of each type they contained.

Another study on Scratch examined the progression of users' programming skills [**?**]. This quantitative analysis of elementary programming skills included measurements of "breadth," the range of different features people used, and "depth," the amount with which people used these features. Scratch's 120 different programming primitives were grouped into 17 categories and the total number of distinct categories of primitives in each project measured its breadth. The total number of primitives per animation measured a project's depth. Our metric of the total number of unique blocks in a project is similar to those used in this study.

An environment that enables users to develop mobile applications directly from their mobile devices is TouchDevelop. A field study of end-user programming on mobile devices was conducted with the objectives including measuring users' progress with developing TouchDevelop scripts [**?**]. Researchers found that 71.3% of users learned a few features about the environment initially and then stopped learning new features. To encourage more continuous learning, researchers suggested providing an adaptive tutoring system that recommends tutorials similar to the kind of script a user is developing and avoids tutorials that cover features users already know. As discussed later in the paper, our findings suggest that App Inventor may also have a similar situation where users tend to only learn a subset of features available to them.

4. Technical Approach

We extracted features from a random sampling of App Inventor 2 projects. We grouped projects based on their functionality by considering the components they contain. We then measured the total number of unique blocks (NOUB) in the projects to determine the intricacy they exhibit. Finally, we examined the distribution of the NOUB in each group to answer our research question.

4.1 Data Source

Our source data is 5,228 App Inventor 2 projects selected at random from the total corpus of 8.3 million projects. We used Pandas, a Python data analysis library, for our data processing [**?**].

Of the 5,228 projects sampled:

- At least 16.4% (859 projects) are recreations of App Inventor tutorials. These recreations of the step-by-step tutorials were identified by matching project names. We considered only the 26 tutorials from the MIT App Inventor website, although many other tutorials made by other groups and individuals exist [**?**][**?**]. Projects that are recreations of these tutorials are filtered out of our dataset.
- 21% (1,107 projects) are certainly static; that is, they are guaranteed to be apps that have no behavior and never change state. If a project has no components, then there is nothing the user can interact with or for the app to do, so the project must be static. For an app to be interactive and have behavior, in addition to at least one component, it must also have at least two blocks: One to handle an event and one to respond to that event. Figure 2 shows an example of a simple action from two blocks. No functionality can occur with fewer than two blocks. We say an app is certainly static if it either has no components or has fewer than two blocks.

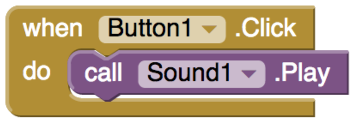


Figure 2. The simplest app behavior requires at least two blocks: An event handler and a resulting action. Here, a sound is played when a button is pressed.

We choose to filter out the certainly static projects as well as projects that are recreations of tutorials, so our analysis was run over the remaining 3,289 projects. While we can guarantee that the removed projects are static, we cannot guarantee the remaining projects have behavior, as their blocks may not be connected in a manner that allows for any behavior. Further improvements to filtering apps are discussed in the conclusion. For the purpose of analysis, we assume the remaining 3,289 projects have behavior and are not recreations of tutorials.

4.2 Feature Extraction

We focused primarily on quantitative features for our analysis, particularly the number of each type of component in a project, and the number of each type of block. This information exists in the source code of the projects.

Features Extracted from Projects:

- Project Name
- Username (anonymized)
- Number of Components by Type
- Number of Blocks by Type

4.3 Grouping Projects

We use the components within a project to group them by functionality. The palette in App Inventor organizes components by functionality, or behavior, and places each group in its own ”drawer” (Figure 3). Because the palette neatly organizes components by functionality into categories , we use it to define our groups. If an app has components from multiple palette drawers, it may be categorized in multiple groups, as explained later in this section.

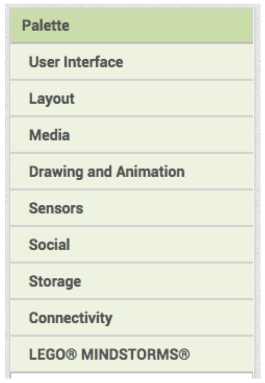


Figure 3. The palette groups components into categories. We use these categories to group projects by functionality.

We follow the palette drawers to define our groups, with two notable changes: Disregarding the entire ”Layout” component drawer and the sound and clock components.

Layout components were removed because they do not add additional functionality and are therefore irrelevant for our groupings. These components only enable users to change the arrangement of an app’s visual components. Our emphasis is to group projects by their functionality, not their appearance or design.

The sound and clock components were removed to improve the differentiation between functionality groups. The sound component plays a sound whenever the user specifies. Examples include playing a ”meow” when an image of a cat is pressed and playing a famous speech in a historical quiz app. The clock component enables apps to keep track of time. Uses for this vary from keeping time in a stopwatch app to periodically moving a sprite in a game app. Because the sound and clock components have such broad uses, they do not help differentiate apps’ behaviors between groups and are also excluded in the consideration of functionality groups.

We categorize the 3,289 apps into eight groups. Basic apps only contain User Interface components. Apps in the Media, Drawing, Sensor, Social, Storage, Connectivity, and Lego groups contain at least one component from that respective drawer in the palette. This categorization allows for overlap, as projects that contain components from multiple palette drawers are placed in multiple groups. For example, a project that uses both Bluetooth (connectivity) and Twitter (social) components would be grouped as both a Connectivity and Social app. The exception is the Lego group, which we deem to be an exclusive group because of the specificity of the components. Lego components are solely for integration with Lego Mindstorms [**?**]; if a project contains a Lego component, it is only grouped as Lego, regardless of other components it may contain.

Reiterating, Basic and Lego groups are disjoint from other groups and each other. Other groups may overlap. Table 1 provides a description of each group, the condition for a project to be in that group, and example apps and components from each group.

We use the components to group projects and the blocks to measure the intricacy of them.

Group Name	Description of App Functionality {Example App}	Condition	Example Components
Basic	Only basic user interface components {Splits bill amongst certain number of people}	Only User Interface Components	Button, Image, Label, Notifier, Textbox
Media	Playing/recording audio or video {Click on picture of politician to hear their famous speech}	At least 1 media component (excluding ”sound”)	Camera, TextToSpeech, VideoPlayer, MusicPlayer
Drawing	Use screen as canvas for drawing {Draw on picture of cat}	At least 1 drawing component	Canvas, Ball, ImageSprite
Sensor	Response to phones’ sensors {Shake phone to roll a die}	At least 1 sensor component (excluding ”clock’)	Accelerometer Sensor, Location Sensor, NearField (NFC)
Social	Communication via phone or web {Click on a persons picture to call or text them}	At least 1 social component	Texting, Twitter, PhoneCall
Storage	Saving information {Add items to grocery list and save list}	At least 1 storage component	TinyDB, Fusiontables Control, File
Connectivity	Networking with other apps and phones {Get latest stock quotes from web}	At least 1 connectivity component	ActivityStarter, Bluetooth-Client, Web
Lego	Control Lego Mindstorm kits {Remote control for Lego Mindstorm NXT robot}	At least 1 lego components	NxtDrive, NxtLight-Sensor

Table 1. Functionality Groupings

4.4 Measuring Programmatic Intricacy

We define the intricacy of an App Inventor project as a measurement of the skill involved to create an app as evidenced by the blocks used. A more intricate app tends to either use more components or use blocks corresponding to these components more effectively.

Code reuse is a particular focus in our measure of intricacy. For example, consider the case where two functionally similar projects exist and Project A copies the same code in three locations whereas Project B defines a procedure and calls that procedure three times. We argue Project B is more intricate as it leverages code reuse in the form of procedures. Project A has a greater number of blocks, but Project B has a greater number of unique blocks with the block to define a procedure and the block to call a procedure included. A project that appropriately uses a procedure rather than copying blocks shows evidence of greater computational thinking and therefore greater intricacy[**?**], even if the resulting apps have identical functionality.

We measure programmatic intricacy of App Inventor projects by looking at the NOUB that exist in the project. We choose the NOUB instead of the total number of blocks so the measure of intricacy is not affected by redundant code. This metric is consistent with previous analysis of Scratch, which has a similar yet simpler scripting language [**?**] [**?**].

5. Results

We show the division of the projects into groups then show the distribution of the number of unique blocks (NOUB) in projects of each group.

5.1 Grouping

After grouping projects by functionality, we find that 78.1% of projects can be categorized into a single group, with the remainder of the projects being categorized into multiple groups. The 3,289 projects were categorized into 4,282 groups; on average, a project fit into 1.3 groups. Figure 4 shows the division of projects into groups.

Based on the distribution of projects into groups we observe, we hypothesize a correlation between this distribution and App Inventor tutorials. Due to the simplicity of functionality that defines the group, the Basic group is the largest. Over half of the App Inventor beginner tutorials involve the creation of a drawing app [**?**] and we see that the Drawing group is the second largest group. These observations suggest that the large number of drawing apps users create are projects that are very similar in functionality to tutorials. The Lego group is the smallest, containing only 0.7% (27 projects) of the data. One likely explanation is the additional hardware requirement (Lego Mindstorm kits) to use an app grouped as Lego. Another is that there are no official tutorials for Lego projects, so users do not have a way to learn how to use the Lego components. We hypothesize that the number of projects in each functionality group correlates with the number of functionally similar tutorials available. We further address this in our Discussion section.

5.2 Number of Unique Blocks

We plot the distribution of the NOUB in each group and compare these subsets of projects to each other and to the entire set of projects. Figure 5 and Table 3 show the NOUB for projects within each group, as well as the distribution for all projects ("All" in Figure 5).

Each subset and the entire set of projects exhibits a positive skew, suggesting that each group contains a few outlier projects that have a significantly greater NOUB and are likely well-developed apps.

The Storage group has the greatest median NOUB, the widest distribution, and contains the project with the most unique blocks, suggesting that apps that utilize storage functionality tend to be the most advanced and intricate. This could be because storage components often require structures such as lists and loops to leverage its more advanced functionality. An example would be using a loop to iterate over the keys and values in a database (TinyDB) component and saving values into a list.

The wide lower quartile and narrow overall distribution of the Lego group suggests its capabilities are more limited. The Lego group has a wide lower quartile (lower whisker in Figure 5) relative to its narrow distribution, suggesting that even a simple project involving Lego components requires more unique blocks to create. The narrow distribution and low median for the Lego group suggests that the capability to create Lego apps is limited. The need for more unique blocks to create even a simple app with Lego components and the limited functionality of these apps suggests that developing these apps is not as intuitive and therefore more difficult.

Because 21.9% of projects fit into multiple groups, one project can be represented in multiple plots. This is most evident in the outliers. The greatest outlier is a password keeper app with 56 unique blocks in it; it is categorized as a Storage, Connectivity, and Media app because it has components of each of those types.

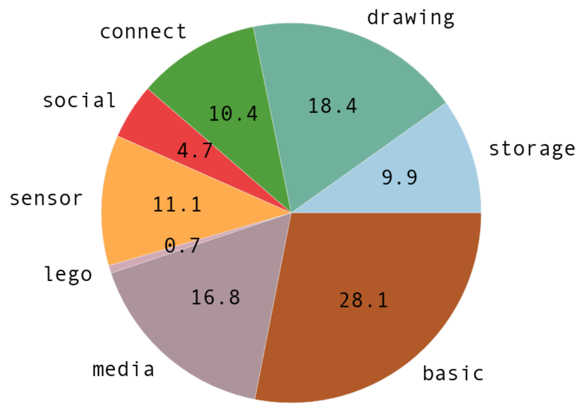


Figure 4. Size of Functionality Groups. 78.1% of projects are categorized into exactly one group, with the others categorized across multiple groups.

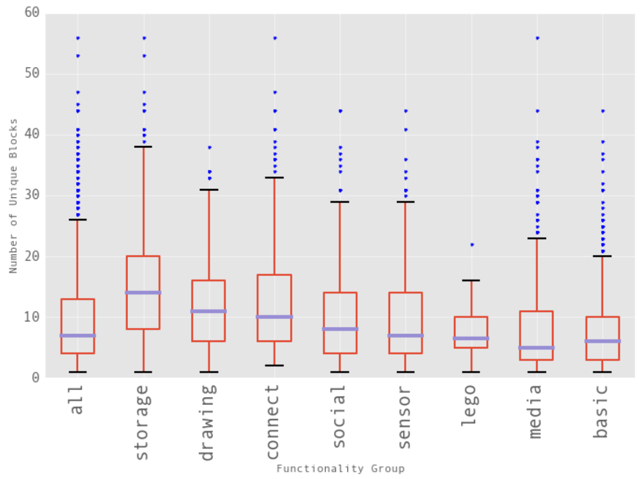


Figure 5. Distribution of Number of Unique Blocks by Functionality Group

6. Discussion

We critique our use of the number of unique blocks as a metric for measuring intricacy and analyze the intuitiveness of creating different types of apps with App Inventor. We then relate this discussion on usability and capability to App Inventor tutorials.

	all	storage	drawing	connect.	social	sensor	lego	media	basic
med.	7	14	11	10	8	7	6.5	5	6
mean	9.17	15.40	11.29	12.34	10.19	9.80	7.86	8.14	9.17
std. dev.	7.11	9.61	6.83	8.79	8.73	7.43	4.86	6.94	5.80
max (w/o outliers)	26.5	38.0	31.0	33.5	29.0	29.0	17.5	23.0	20.5
# outliers	90	11	6	14	10	12	1	26	41

Table 2. Summary Statistics for the Number of Unique Blocks by Group

6.1 Analysis of Metric

When measuring the intricacy of projects, our challenge is to ensure that project categories do not bias our metrics. That is, our measurement of app intricacy is not affected simply because apps include a specific component and hence fall under a certain group. We want to measure apps solely according to the intricacy exhibited by the blocks. We argue that our metric of the NOUB is not dependent on the functionality of the app and is therefore a generalizable metric of programmatic intricacy.

Because App Inventor a custom block for each functionality of a component, the NOUB in a project is not directly dependent on its components. App Inventor is event-driven, meaning the programming of App Inventor involves responding to an action, or event, from a component. Each component has its own unique blocks to handle events, get and set attributes of the component, and call component functions. Because of this, using one component instead of another does not inherently change the NOUB in a project. App Inventor blocks respond to events, get/set attributes, and trigger component actions. Because of App Inventor's component-specific blocks, the NOUB in a project is a suitable metric to measure the intricacy of projects.

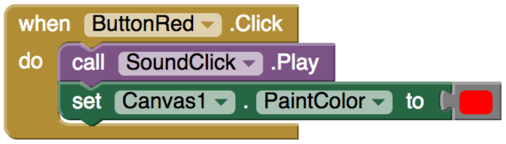


Figure 6. Component-Specific Blocks. The button component has a block to handle being pressed, the sound component has a block to play a sound, and the canvas component has block to set the color.

Because the NOUB does not systematically vary according to the components used in the projects, we find this metric suitable for our analysis.

6.2 Considering Control Constructs

Another metric used in previous research with blocks-based languages for measuring programmatic skill is the measuring the number of "control constructs" evident in a project [?]. To measure the existence of control constructs in the context of App Inventor, we would specifically assess the number loops, lists, conditionals, procedures, and/or variables used in apps with different functionality. This metric was considered but we find that it is too dependent on the functionality of the app to be used. For example, Storage apps frequently utilize lists as temporary storage between the app and the database, whereas drawing apps typically involve a canvas for the user to draw on and rarely have a purpose for lists. Because our focus is on comparing different functionality groups, measuring the number of specific control constructs is not appropriate because different constructs lends themselves towards different functionalities.

6.3 Usability

We define a group to have high usability if it does not require many different blocks to create a simple project. If a group has high usability, we expect many projects to be categorized into that group. We define a "usability score" of a group as the number of projects in the group divided by the the median number of unique blocks for that group. The results for the different groups are depicted in Figure 7.

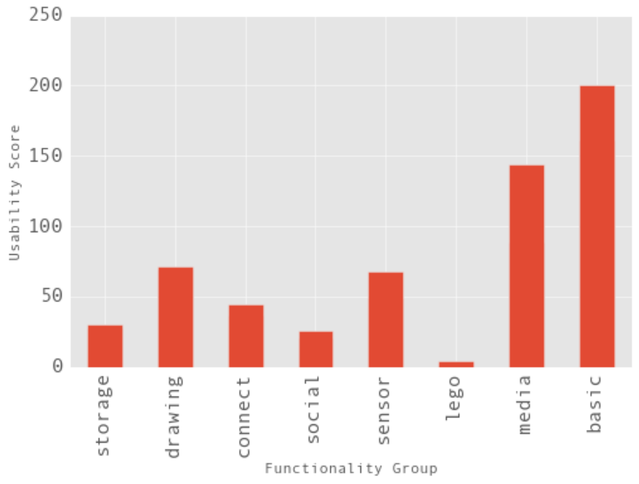


Figure 7. Usability Score (Ratio of the Number of Projects to Median Number of Unique Blocks) of Functionality Groups

Apps in the Basic group have the highest usability score and are therefore the easiest to learn. This is not surprising because we narrowly define the Basic group to contain apps that only use user interface components. The Drawing and Media groups also have high usability scores. This is likely because the tutorials heavily focus on creating Drawing apps and Media apps. We argue that the usability score is influenced by the beginner tutorials and address this later in the Discussion section.

6.4 Capability

We focus our discussion on realized capability, or the maximum potential of App Inventor that users actually reach. We are not referring to the "true" capability of App Inventor, the capability that is technically possible but in practice almost never implemented by users.

To assess the realized capabilities for App inventor to create apps of certain functionalities, we are interested in the projects in each group that have the greatest intricacy (greatest NOUB). It is these projects that best reflect the capability of App Inventor to accomplish certain functionality. We choose to look at the the maximum NOUB in each group excluding outlier projects. This (non-outlier) maximum is the end of the upper whiskers in Figure

5 and also shown in Table 3. We argue that these apps best represent the capability realized by App Inventor users.

A greater maximum NOUB correlates with the ability to use App Inventor to create apps that have more advanced functionality. Having the least capability are Lego apps, with the narrowest distribution and lowest maximum NOUB. On the highest end of the spectrum are Storage apps which leverage databases or tables to persist data.

We see that apps with the greatest capability tend to connect and extend the app to other functionality on a mobile device or with the web. Storage and connectivity apps have the greatest maximum NOUB so we say these groups have the greatest realized capability. Storage apps connect to some form of data persistence (database, table, file). Connectivity apps connect with other apps on the phone, utilize Bluetooth, and connect with the web APIs. What is interesting is that Social apps connect to other features in the phone (contact list, texting, etc.) or with social media such as Twitter, yet their realized capabilities are lower. This could be a result of a lack of learning resources relating to this particular functionality limiting the known capability of the group, which we discuss next. In general though, we see that the apps with the most realized capability tend to connect to the web and other apps and services. This opportunity for extensibility while maintaining the scaffolding that is the App Inventor environment is critical to an environment that fosters computational thinking [**?**].

6.5 Relation to Learning Resources

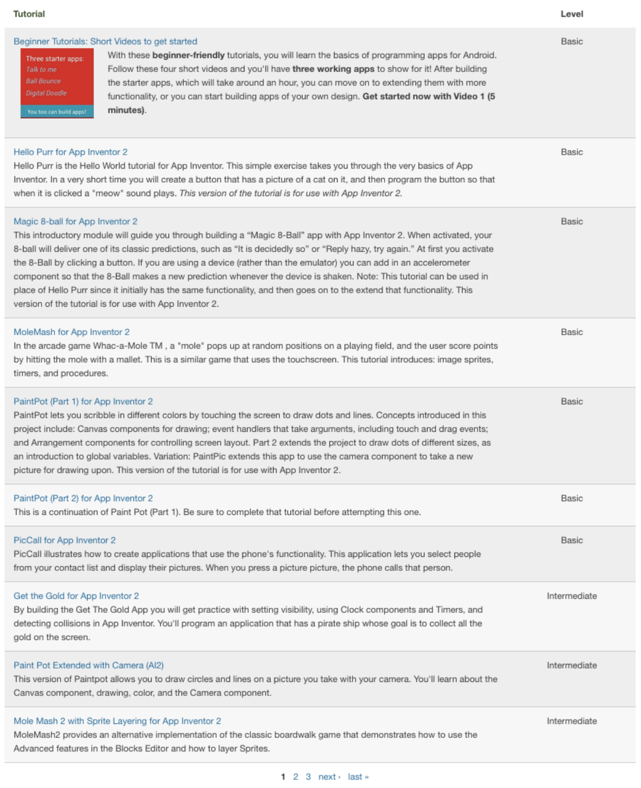
The close correlation between usability scores and the order of App Inventor tutorials suggests that users build apps based on knowledge from the tutorials that they complete. On the App Inventor website, tutorials are displayed as a list in sequential order, starting with Beginner tutorials and ending with Advanced tutorials. Table 4 shows the number of projects that were found to be tutorial recreations as well as the number of tutorials for each group.

Most users tend to start with the beginner tutorials, but the number of tutorials followed until users create their own original projects varies. The earlier a tutorial appears in the sequence on the website, the more users will use it. We see Drawing, Media and Sensor apps appear as beginner tutorials; these groups also account for most of the tutorial recreations and have the highest usability scores. There are no Lego tutorials, and the very low usability score reflects that. Although there are six tutorials involving Storage functionality, they are classified as Intermediate and Advanced, so there are fewer recreations of these tutorials. The Connectivity and Social groups also have a low usability score and few projects recreate these tutorials. We observe that lower usability scores correlate to groups that have fewer tutorial recreations; the farther along a tutorial exists in the sequence, the fewer times it will be recreated.

		Tutorial Difficulty			
		Beginner	Intermediate	Advanced	Total
Groups	All	683 {9}	95 {10 }	54 {7}	832 {26}
	Storage	0 {0}	5 {1}	48 {5}	53 {6}
	Connectivity	0 {0}	9 {1}	26 {2}	35 {3}
	Drawing	447 {4}	70 {5}	5 {2}	522 {11}
	Social	0 {0}	6 {1}	0 {0}	6 {1}
	Sensor	142 {2}	0 {0}	35 {4}	177 {6}
	Lego	0 {0}	0 {0}	0 {0}	0 {0}
	Media	198 {2}	4 {3}	0 {0}	202 {5}
	Basic	81 {1}	11 {1}	0 {0}	92 {2}

Table 3. Number of Apps that are Tutorial Recreations {Number of Tutorials} by Functionality Group and Difficulty Level

This correlation between the number of tutorial recreations and the usability scores for a given functionality suggests that users



Tutorial	Level
Beginner Tutorials: Short Videos to get started Three starter apps: *Talk to me* *Ball Bounce* *Digital Doodle* You too can build apps! With these **beginner-friendly** tutorials, you will learn the basics of programming apps for Android. Follow these four short videos and you'll have **three working apps** to show for it! After building the starter apps, which will take around an hour, you can move on to extending them with more functionality, or you can start building apps of your own design. **Get started now with Video 1 (5 minutes)**.	Basic
Hello Purr for App Inventor 2 Hello Purr is the Hello World tutorial for App Inventor. This simple exercise takes you through the very basics of App Inventor. In a very short time you will create a button that has a picture of a cat on it, and then program the button so that when it is clicked a "meow" sound plays. *This version of the tutorial is for use with App Inventor 2.*	Basic
Magic 8-ball for App Inventor 2 This introductory module will guide you through building a “Magic 8-Ball” app with App Inventor 2. When activated, your 8-ball will deliver one of its classic predictions, such as “It is decidedly so” or “Reply hazy, try again.” At first you activate the 8-Ball by clicking a button. If you are using a device (rather than the emulator) you can add in an accelerometer component so that the 8-Ball makes a new prediction whenever the device is shaken. Note: This tutorial can be used in place of Hello Purr since it initially has the same functionality, and then goes on to the extend that functionality. This version of the tutorial is for use with App Inventor 2.	Basic
MoleMash for App Inventor 2 In the arcade game Whac-a-Mole TM , a "mole" pops up at random positions on a playing field, and the user score points by hitting the mole with a mallet. This is a similar game that uses the touchscreen. This tutorial introduces: image sprites, timers, and procedures.	Basic
PaintPot (Part 1) for App Inventor 2 PaintPot lets you scribble in different colors by touching the screen to draw dots and lines. Concepts introduced in this project include: Canvas components for drawing; event handlers that take arguments, including touch and drag events; and Arrangement components for controlling screen layout. Part 2 extends the project to draw dots of different sizes, as an introduction to global variables. Variation: PaintPic extends this app to use the camera component to take a new picture for drawing upon. This version of the tutorial is for use with App Inventor 2.	Basic
PaintPot (Part 2) for App Inventor 2 This is a continuation of Paint Pot (Part 1). Be sure to complete that tutorial before attempting this one.	Basic
PicCall for App Inventor 2 PicCall illustrates how to create applications that use the phone's functionality. This application lets you select people from your contact list and display their pictures. When you press a picture picture, the phone calls that person.	Basic
Get the Gold for App Inventor 2 By building the Get The Gold App you will get practice with setting visibility, using Clock components and Timers, and detecting collisions in App Inventor. You'll program an application that has a pirate ship whose goal is to collect all the gold on the screen.	Intermediate
Paint Pot Extended with Camera (AI2) This version of Paintpot allows you to draw circles and lines on a picture you take with your camera. You'll learn about the Canvas component, drawing, color, and the Camera component.	Intermediate
Mole Mash 2 with Sprite Layering for App Inventor 2 MoleMash2 provides an alternative implementation of the classic boardwalk game that demonstrates how to use the Advanced features in the Blocks Editor and how to layer Sprites.	Intermediate

Figure 8. App Inventor tutorials are displayed in sequential order despite the fact that content of tutorials often do not build off each other.

build off the knowledge from tutorials when creating an app. The relationship between few tutorial recreations for groups and low usability scores suggests that if users do not learn a concept directly from a tutorial, they tend to have trouble generalizing knowledge from other tutorials. Therefore, users tend not to create apps that are functionally different from completed tutorials. This is troublesome as [**?**] noted that 71.3% of users of the TouchDevelop environment tended to learn only a few features initially and not seek to learn more later. We observe that users do not complete enough tutorials to gain a thorough understanding to create apps of differing functionality with App Inventor, so there exists a need to make tutorials more available and the knowledge from them more generalizable.

To better prepare users to create apps with more diverse functionality, we propose changes to the App Inventor learning resources to ensure tutorials are more accessible and contain more transferrable knowledge. We consider the following changes to App Inventor tutorials:

- Avoid Pre-defined Paths for Tutorials: As of now, App Inventor tutorials exist as a list where tutorials often do not build off of knowledge from previous ones. So, we suggest that tutorials be offered in such a way that users do not feel obligated to follow an unnecessary predefined ”path” when recreating tutorials. Instead, users should be more inclined to select tutorials relevant to their interests.
- Organizing Learning Resources: Because learning resources tend to be separate from each other on the MIT App Inventor website, users may not know where to go when they encounter a problem, or they may not even know given resources exist. For example, concept cards, which explain specific concepts and do not teach the creation of full apps, can serve as quick references for users [**?**]. They are placed with the teaching

resources on the App Inventor website, entirely separate from tutorials. Ensuring that users have a centralized and organized point to access resources would better support users.

- Modularize Tutorials to Focus on Functionality: App Inventor tutorials focus on how to develop a functioning app. The need we find on the forums is help on how to accomplish a specific functionality, such as persisting a high score in a game or sharing data across multiple screens aiForum. Having a 5-10 minute tutorial on ”How to use Lists” rather than an hour tutorial that records a list of addresses and views them on Google Maps (”Map It: Displaying Locations on a Google Map,” the only tutorial with lists), enables users to succinctly learn the specific functionality they are in question about.
- Leverage the Community Gallery: The App Inventor gallery is a recent addition to App Inventor which enables users to share their projects and ”remix” and build off of projects other users shared aiGallery. With this, App Inventor learning resources no longer need to show the creation of apps from blank, completely new projects. Instead, they can look to well-built apps shared by the community and highlight ones that other users can learn from and build off of.
- Adaptive Tutorials: Suggested in a similar context by [**?**], an adaptive tutoring system would recommend resources relevant to what the user is creating and avoid concepts already learned or implemented. This would enable users to monitor their own progress and give them a more holistic perspective of the functionalities offered with App Inventor environment.

7. Conclusion

In this paper, we evaluate the usability (low floor) and capability (high ceiling) of App Inventor, a web-based environment that enables users to create mobile apps with a visual blocks based programming language. From our sample of 5,228 apps, we filtered out certainly static apps and apps that are recreations of our tutorials and grouped apps by similar functionality. We then measured the number of unique blocks for projects in each group and compared the groups.

Our critical findings: (1) there exists a strong correlation between the usability and the number of tutorial recreations for a given functionality; (2) users tend to follow tutorials sequentially and therefore often do not complete more than beginner tutorials; (3) users tend to develop apps that are functionally similar to completed tutorials; (4) apps with the greatest realized capability tend to connect to other functionality on a mobile device or with the web (external databases, APIs, etc.). Based on these findings, we suggest that the realized capabilities of App Inventor are limited by the provided learning resources. We provide a list of recommendations for improving these resources for end-users. These recommendations are not specific the App Inventor environment and are transferable and applicable to other programming environments and tools that have online resources.

The existence of non-functional projects and recreations of tutorials in our dataset offer opportunities for future work. While we filtered out projects that were certainly static by ensuring all projects had the minimum number of components and blocks, there still exist projects in our dataset that do not have any functionality. Disregarding blocks that are not connected to other blocks and components with no programmed functionality would better filter out the non-functional projects. And although we filter out projects that are recreations of tutorials by matching project names, a more robust method of identifying projects that are merely recreations would improve the dataset. We could also focus analysis on specific advanced structures more closely tied with computational thinking such as iterators and procedures, as defined by [**?**]. Finally, there is promise in analyzing user and temporal data, investigating how users and their apps develop over time.

Acknowledgments

We thank Jeffery Schiller (MIT App Inventor) for helping collect the data, Ilaria Liccardi (MIT) and Franklyn Turbak (Wellesley College) for helping guide the analysis, and Aubrey Colter (MIT App Inventor) for significantly helping proofread this paper.

This research is funded by the MIT EECS - Google Research and Innovation Scholarship as part of the 2014-15 MIT SuperUROP Program.

References

[1] P. Q. Smith, and X. Y. Jones. ...reference text...